Argtable Reference Manual

version 1.3

Generated by Doxygen 1.2.6

Fri Dec 21 00:12:19 2001

Contents

1 Introduction to Argtable 1

2 Argtable Data Structure Index 3

3 Argtable File Index 3

4 Argtable Page Index 4

5 Argtable Class Documentation 4

6 Argtable File Documentation 6

7 Argtable Page Documentation 10

1 Introduction to Argtable

1.1 Legal notice.

1.2 Overview.

Argtable (http://argtable.sourceforge.net) is a freely available programmer's library for parsing the command line arguments of any C/C++ program. It allows the programmer to specify the desired format of the command line arguments in one or more statically defined arrays known as argument tables. Each row of an argument table specifies the data type of an expected argument and nominates a user-defined program variable as storage for the incoming argument value. If arguments require default values, then these too are specified in the argument table.

Once an argument table has been established, parsing the command line is simply a matter of calling the library's arg_scanargv() function which attempts to resolve the contents of **argv[ ]** with the entries of the argument table. If successful, the command line arguments are now known to be valid and their values are ready and available for use in their nominated program variables.

If the arguments could not be successfully resolved then **arg_scanargv** returns an error message string describing the reason for the failure and showing the location of the error in the command line. The program can simply print the error message to stdout or stderr and exit.

```
ERROR: myprog grad:13 99 uh oh
                         ^^ unexpected argument
```

Alternatively, if the program has multiple command line usages then it may choose to call **arg_scanargv** several times each with a different argument table until a successful match is found or all argument tables are exhausted.

Auxilliary functions arg_glossary() and arg_syntax() generate plain text descriptions of the arguments defined in an argument table and their command line syntax. These make it easy to generate on-line help facilities that are always current.

1.3 Styles of command line arguments.

Argtable supports both *tagged* and *untagged* command line arguments. Tagged arguments are identified by a prefix tag, as in **-o file** or **title:mystuff**. The tag enables these arguments to appear anywhere on the command line, and in any order. The programmer may implement any style of argument tag desired, including such common styles as **-title mystuff**, **title:mystuff**, **–title mystuff**, or **title=mystuff**. Untagged arguments on the other hand have no prefix; they are identified strictly by their ordering on the command line. The two styles of arguments may be freely mixed, whereupon the tagged arguments are always processed first, leaving any remaining (untagged) arguments to be scanned from left to right.

A command line argument may be of type **integer**, **double**, **string**, or **boolean**. Doubles are accepted in in either floating point or scientific notation, and strings may be either quoted or unquoted. Booleans will accept any of the keywords **on**, **off**, **yes**, **no**, **true**, or **false** and yield an integer value of 0 (negative) or 1 (affirmative) accordingly.

A special argument type called **literal** is also provided; it yields an integer value according to the presence or absence of a given string literal on the command line. It is useful for specifying unparameterised command line switches such as **-verbose** and **-help**.

1.4 Optional arguments and default values.

Arguments may be assigned default values that take effect when no matching command line argument could be found. When a default value is specified for an argument you are, in effect, declaring that argument as being optional. Arguments without defaults are, by definition, regarded as mandatory arguments.

1.5 Supported platforms.

Argtable conforms to ansi C requirements and should compile on any standard ansi C compiler. To date, it has been successfully compiled on:

- MIPSpro C/C++ on IRIX 6.2, 6.3 and IRIX64 6.2
- DEC C/C++ on Digital Unix V4.0 (OSF/1)
- GNU gcc/g++ on DEC Digital Unix V4.0 (OSF/1); IRIX 6.2, 6.3; IRIX64 6.2; Linux 2.0.30; and SunOS 5.5.1.

Please let me know if you have successfully used argtable on other platforms.

1.6 Installing Argtable

The fastest and easiest way to use argtable is simply to copy the **argtable.h** and **argtable.c** files into your project and compile them with the rest of your code.

If you are a system administrator, you may wish to install argtable on your system as a programmer's library, complete with man pages and html documentation. This is easy to do as the makefiles follow the usual **autoconf** procedure.

1. **cd** to the argtable package directory and type **./configure** to configure it for your system.
2. Type **make** to compile the package.
3. Optionally, type **make check** to run the self-tests.
4. **su** as root user.
5. Type **make install** to install the programs and documentation. By default the package installs into **/usr/local/** but a different location may be specified when configuring the package. Type **./configure –help** for the full list of configuration options.
6. logout from root user.
7. Type **make clean** to remove the temporary binaries created during compilation.

1.7 Getting started.

The Argtable Tutorial is the best place to get started. Then look at the example code supplied with the distribution.

1.8 Similar packages.

Here are some other command line parsing tools that I am aware of. Apologies for any I may have omitted.

- **clig**: The Command Line Interpreter Generator: `http://wsd.iitb.fhg.de/~kir/clighome/`
- **opt**: The Options and Parameter parsing Toolkit: `ftp://ftp.lanl.gov/pub/users/jt/Software/opt/opt_toc.html`
- **getopt**: The GNU-style options parser. This comes as standard on most unix systems.

2 Argtable Data Structure Index

2.1 Argtable Data Structures

Here are the data structures with brief descriptions:

arg_rec 4

3 Argtable File Index

3.1 Argtable File List

Here is a list of all files with brief descriptions:

argtable.h 6

4 Argtable Page Index

4.1 Argtable Related Pages

Here is a list of all related documentation pages:

The Argtable Tutorial 10

Release Notes. 15

5 Argtable Class Documentation

5.1 arg_rec Struct Reference

`#include <argtable.h>`

Data Fields

- const char* tagstr
- const char* argname
- arg_type argtype
- void* valueptr
- const char* defaultstr
- const char* argdescrip

5.1.1 Detailed Description

An argument table is defined an array of arg_rec structs having one entry for each command line argument that is expected. It is most conveniently defined and initialised as a static array (but need not be so).

```
//command line arguments will be written into these variables
int x,y,z,verbose,debug;
double radius;
char infname[100];
char outfname[100];i

//The argument table.
arg_rec argtable[] =
  // TAG        NAME         TYPE     STORAGE   DEFAULT DESCRIPTION
  {
  { NULL,       "x",         arg_int,  &x,       NULL,  "x coord" },
  { NULL,       "y",         arg_int,  &y,       NULL,  "y coord" },
  { NULL,       "z",         arg_int,  &z,       "0",   "z coord" },
  { "-r ",      NULL,        arg_dbl,  &radius,  "1.0", "radius" },
  { "-o ",      "<outfile>", arg_str,  outfname, "-",   "output file" },
  { "-verbose", NULL,        arg_lit,  &verbose, "0",   "verbose output" },
  { NULL,       "<infile>",  arg_str,  infname,  NULL,  "input file" },
  { "debug=",   "<on/off>",  arg_bool, &debug,   "off", NULL },
  };
const size_t narg = sizeof(argtable)/sizeof(arg_rec);
```

5.1.2 Field Documentation

5.1.2.1 const char * arg_rec::tagstr

The argument tag string defines the tag that identifies the argument on the command line. Tags may be any string you choose provided they have no whitespace. Examples of common tag string styles are ”mytag:”, ”-mytag”, ”mytag=”. An argument can be specified without a tag by setting the tag string to NULL. Untagged argument are taken from the command line from left to right after all the tagged arguments (if any) have first been extracted.

5.1.2.2 const char * arg_rec::argname

The argument name has no effect on the argument processing. It is simply a descriptive name used to represent the argument in the output of the arg_syntax() and arg_glossary() functions. The argument name can be whatever you want, and is a convenient place to communicate the default value to the user if you so wish, for example ”<size>=1024”. If a NULL name string is given, it is automatically replaced by the argument’s data type enclosed in angled brackets, as in ”<integer>”. If you dislike such behaviour, you can effectively suppress the name by defining it as an empty string.

5.1.2.3 arg_type arg_rec::argtype

This defines the data type associated with a command line argument. It supports integer, double, string and boolean data types as well as literal argument strings.

Strings may appear on the command line either quoted or unquoted.

Booleans expect one of the keywords ”true”, ”false”, ”yes”, ”no”, ”on”, or ”off” to appear on the command line. These are converted to 0 (false,no,off) or 1 (true,yes,on) and stored as an integer.

Literals are command line arguments with no associated data value, they are used to define keyword strings that can be used as command line switches. The string literal can be defined in either the tag string or the name string fields. If you use the tag string then the literal, like other tagged arguments, may appear anywhere on the command line. On the other hand, if you use the name string, then the literal must appear in that argument’s position just as for normal untagged arguments. When a string literal is succesfully scanned, an integer value of 1 is written into its user-suplied program variable, otherwise it is assigned the default value (if it has one). If there is no default value, then the literal is regarded as a mandatory argument the same as for any other argument.

5.1.2.4 void * arg_rec::valueptr

Points to a user-defined variable into which the command line value will be written. It is imperative that the data type of the user-defined variable matches the argtype field, otherwise you can expect very spurious behaviour.

- *arg_int* arguments require *int* storage.
- *arg_double* arguments require *double* storage.
- *arg_str* arguments require a *char* buffer big enough to store the expected result (you’ll have to guess what big enough is!).
- *arg_bool* arguments require *int* storage.
- *arg_lit* arguments require *int* storage.

Lastly, a NULL placed in this field will cause the argument value to be scanned in the normal way, but the resulting value will be discarded.

5.1.2.5 const char * arg_rec::defaultstr

The default string contains an optional default value to be used should the argument be missing from teh command line. All defaults are defined as strings (as they would appear on teh command line) and converted to the appropriate data type during processing.

If a default is specified as NULL then that argument is regarded as mandatory, meaning that a parse error will result if the argument was not given on the command line.

5.1.2.6 const char * arg_rec::argdescrip

The argument description string, like the name string, does not affect argument processing. The arg_glossary() function uses it to display additional descriptions of command line arguments. Arguments with NULL description strings are omitted from the glossary altogether.

6 Argtable File Documentation

6.1 argtable.h File Reference

```
#include <stdio.h>
```

Data Structures

- struct arg_rec

Defines

- #define ARGTABLE_VERSION 1.3

Enumerations

- enum arg_type { arg_int = 0, arg_dbl, arg_str, arg_bool, arg_lit }

Functions

- int arg_scanargv (int argc, char **argv, arg_rec *argtable, int n, char *CmdLine, char *ErrMsg, char *ErrMark)

  Parse the command line as per a given argument table.

- int arg_scanstr (char *str, arg_rec *argtable, int n, char *ErrMsg, char *ErrMark)

  Parse a string as per a given argument table.

- const char* arg_syntax (const arg_rec *argtable, int n)

  Generates a 'usage' syntax string from an argument table.

- const char* arg_glossary (const arg_rec *argtable, int n, const char *prefix)

  Generate a glossary string from an argument table.

- void arg_catargs (int argc, char **argv, char *str)

Concatenate all argv[] arguments into a single string.

- arg_rec arg_record (char *tagstr, char *argname, arg_type argtype, void *valueptr, char *defaultstr, char *argdescrip)

  Builds and returns an argument table record.

- void arg_dump (FILE *fp, const arg_rec *argtable, int n)

  Print the contents of an argument table.

Variables

- const char* arg_typestr [ ]

6.1.1 Define Documentation

6.1.1.1 #define ARGTABLE_VERSION 1.3

6.1.2 Enumeration Type Documentation

6.1.2.1 enum arg_type

arg_type enums are used in the argument table to define the data type of a command line argument.

Enumeration values:

arg_int Integer value.

arg_dbl Double value.

arg_str Ascii string; may be quoted or un-quoted.

arg_bool Boolean; accepts the keywords *yes*, *no*, *true*, *false*, *on* , or *off* and converts them into an integer value of 0 (negative) or 1 (affirmative) accordingly.

arg_lit Literal; returns 1 if a given literal string was present on the command line otherwise returns the default value.

6.1.3 Function Documentation

6.1.3.1 int arg_scanargv (int *argc*, char ** *argv*, arg_rec * *argtable*, int *n*, char * *CmdLine*, char * *ErrMsg*, char * *ErrMark*)

Attempts to resolve the argv[] command line arguments (ignoring argv[0]) with the specifications given in the argument table. The values scanned from the command line are written directly into the program variables nominated by each argument table entry.

During the process, a copy of the command line is written (as a single line of space separated arguments) into the user-supplied string at *CmdLine in case it is needed in future for error reporting.

Should there be any conflict between the command line arguments and the argument table specifications, an error message and corresponding error marker are written into the user-supplied strings at *ErrMsg and *ErrMark respectively, after which the function returns zero. The error marker string is used to store a string of tilde characters formated in such a way that the tildes underscore the exact location of the error in *CmdLine when the strings are aligned one above the other. This can be useful for including in on-line error messages to help the user quickly pinpoint the cause of the error.

If, on the other hand, all arguments were resolved successfully then *ErrMsg and *ErrMark are set to empty strings and the function returns 1. Either way, CmdLine, ErrMsg, or ErrMark can be safely ignored by passing them as NULL.

Returns:
1 upon success, 0 upon failure.

Parameters:
argc number of entries in argv[].

argv command line arguments.

argtable pointer to the argument table.

n number of entries in argtable[].

CmdLine pointer to storage for command line (may be NULL).

ErrMsg pointer to storage for error message (may be NULL).

ErrMark pointer to storage for error marker (may be NULL).

6.1.3.2 int arg_scanstr (char * *str*, arg_rec * *argtable*, int *n*, char * *ErrMsg*, char * *ErrMark*)

This function is much like arg_scanargv() except that is scans the arguments from the string at *str rather than from argv[]. The string is expected to contain a single line, space separated list of arguments, like that generated by arg_catargs().

In a departure from **arg_scanargv**, this function erases the scanned arguments from **str* by overwriting them with spaces once they have been successfully scanned. Furthermore, this function does not throw an error if there are still arguments remaining in *str after the argtable has been fully processed. Thus, complicated argument usages can be achieved by invoking this function multiple times on the same command line string, each time applying a different argument table until the arguments have been exhausted, or an error has been detected.

Returns:
1 upon success, 0 upon failure.

Parameters:
str pointer to command line string.

argtable pointer to the argument table.

n number of array elements in argtable[].

ErrMsg pointer to storage for error message (may be NULL).

ErrMark pointer to storage for error marker (may be NULL).

6.1.3.3 const char * arg_syntax (const arg_rec * *argtable*, int *n*)

Builds a syntactical description of the allowable command line arguments specified by the 'argtable' array. The resulting string is stored in static data within the local scope of this function. Its contents are overwritten by subsequent calls.

The syntactical description is generated as a single line of space separated argument descriptors, each comprising of the argument's tag string and name string concatenated together. For example,

"myprog x y [z] [-r <double>] [-o <outfile>] [-verbose] <infile> [debug=<on/off>]"

If an argument is optional (has a non-NULL default value) then its descriptor is enclosed in square brackets. NULL name strings are substituted with the argument's data type enclosed in angled brackets, as in <int>, <double>, or <string>. If both the tag and the name are empty strings ("") then the argument is omitted from the description altogether. This allows the suppression of individual arguments that you do not want to appear.

Returns:
a pointer to the internal string.

Parameters:
argtable pointer to the argument table
n number of array elements in argtable[]

6.1.3.4 const char * arg_glossary (const arg_rec * *argtable*, int *n*, const char * *prefix*)

Returns a pointer to an internal 'glossary' string which contains a multi-line description of each of the argument table entres that have a non-NULL <description> field. The contents of the glossary string remain unaltered up until the next invocation of this function. Each line of the glossary string is formatted as

"<prefix><tag><name><description>"

The 'prefix' string is useful for adding indenting spaces before each line in the description to improve the look of the glossary string, or it can be given as NULL in which case it is ignored.

Any NULL <tag> fields in the argument table will appear in the glosssary as empty strings.

Any NULL <name> fields will be substituted by a description of that argument's data type, enclosed in angled brackets, as in <int> and <double>. A name can effectively be suppressed from the glossary by defining it as an empty string in the argument table.

Returns:
a pointer to the internal string.

Parameters:
argtable pointer to the argument table
n number of array elements in argtable[]
prefix a string to be prefixed to each line of the output

6.1.3.5 void arg_catargs (int *argc*, char ** *argv*, char * *str*)

Concatenates all of the arguments in the argv[] array and writes the result into *str as a single line, space separated string.

Any argv[] entries that contain whitespace are automatically encapsulated by single quotes prior to the concatenation to preserve their word grouping. A trailing space is always appended to the resulting string as a safety precaution in lieu of scanning for string literals that expect trailing space. It is assumed that *str is big enough to store the result.

Parameters:
argc number of arguments in argv[]
argv command line arguments
str pointer to destination string

6.1.3.6 arg_rec arg_record (char * *tagstr*, char * *argname*, arg_type *argtype*, void * *valueptr*, char * *defaultstr*, char * *argdescrip*)

Returns an arg_rec structure containing the values passed to the function. It is useful for building argument tables dynamically.

Parameters:

tagstr argument tag string

argname argument name string

argtype argument data type

valueptr pointer to user-supplied storage location

defaultstr default argument value, as a string

argdescrip argument description string

6.1.3.7 void arg_dump (FILE * *fp*, const arg_rec * *argtable*, int *n*)

The contents of the argument table, and the user-supplied variables it references, are printed to the stream 'fp'. This can be useful for debugging argument tables.

Parameters:

fp output stream

argtable pointer to the argument table

n number of array elements in argtable[]

6.1.4 Variable Documentation

6.1.4.1 const char * arg_typestr

A fixed array of strings that are used when arguments are given NULL names. The array is indexed by arg_type, with each name describing the corresponding data type.

```
arg_str[arg_int]  = "<int>";
arg_str[arg_dbl]  = "<double>";
arg_str[arg_str]  = "<string>";
arg_str[arg_bool] = "<bool>";
arg_str[arg_lit]  = "";
```

7 Argtable Page Documentation

7.1 The Argtable Tutorial

Imagine we have written a program called **myprog** and we wish to implement the following command line usage syntax for it:

```
myprog [-tit <title>] grad:gradient [y-int]
   -tit <title>    your title
   grad:gradient   line gradient
   y-int           y-intercept
```

We will create a single argument table in our program that defines the argument properties, and pass that table along with **argc** and **argv[ ]** to the arg_scanargv() function to do the parsing.

7.2 Defining the argument table.

An argument table is just an array of **arg_rec** structs, with each array element pertaining to a single command line argument. The arg_rec struct is defined in argtable.h as:

```
typedef enum {arg_int=0,arg_dbl,arg_str,arg_bool,arg_lit} arg_type;
typedef struct
   {
   const char *tagstr;       // argument tag string
   const char *argname;      // argument name string
   arg_type    argtype;      // argument data type
   void       *valueptr;     // ptr to user storage location
   const char *defaultstr;   // default value, as a string
   const char *argdescrip;   // argument description string
   } arg_rec;
```

Thus we may define our argument table statically in the code as follows:

```
int main(int argc, char **argv)
   {
   static char str[50];
   static double grad;
   static int c;
   arg_rec argtable[] =
      {
      {"-tit ", "<title>",  arg_str, str,   "noname", "\t\t your title"},
      {"grad:", "gradient", arg_dbl, &grad, NULL,     "\t line gradient"},
      {NULL,    "y-int",    arg_int, &c,    "0",      "\t\t y-intercept"}
      };
   const size_t narg = sizeof(argtable)/sizeof(arg_rec);
   ...
  }
```

Defining the tables statically is a programming convenience but not a requirement; the table could equally well have been dynamically allocated and initialized at runtime. Notice that I also chose to define the argument table within the main() block because that's the only place where it is used so there is no need to promote it to a higher namespace. However you may define it in the global namespace if you prefer.

Our argument table has three rows, one for each command line argument **-tit** <**title**>, **grad:gradient**, and **y-int**. The behaviour of the argument parsing is governed entirely by the contents of the various fields (columns) of the argument table. Lets step through each field one by one.

The tag string:

The first field is the argument's tag string. It defines the prefix literal that identifies a tagged argument and should contain at least one non-whitespace character unless the argument is untagged whereupon the field should be NULL. In our example, **-tit** <**title**> and **grad:gradient** are tagged arguments but **y-int** is untagged so it has a NULL tag string.

The name string:

The second field is the argument's name string. It is not actually used to process the command line arguments, rather it defines the name of the argument as it appears in the description strings generated by the **arg_syntax** and **arg_glossary** functions. Those functions automatically substitute any NULL names with the argument's data type enclosed in angled brackets, as in "<int>" or "<string>".

The data type:

The third field is an enumerated type that defines the data type of the command line argument. Possible values are **arg_int**, **arg_dbl**, **arg_str**, **arg_bool**, and **arg_lit**. They represent integer, double, string, boolean, and literal arguments respectively. In our example **-tit** <**title**> expects <*title*> to be substituted by a string value, **grad:gradient** expects *gradient* to be a double, and **y-int** is expected to be an integer.

The data pointer:

The fourth field is a pointer-to-void that gives the address of the user-defined program variable used to store the argument's value. A NULL pointer here causes the value to be discarded once is has been scanned. Take care that the data type of of the target memory location matches that specified in the previous column. Arguments of type **arg_int**, **arg_bool**, and **arg_lit** must each point to an *integer* variable. Those of type **arg_dbl** must point to a *double* and those of **arg_str** must point to a *char array*. In our example, the string value associated with **-tit** <**title**> is written into the *char str[50]* buffer, the double value associated with **grad:gradient** is written into *double grad*, and the integer value associated with **y-int** is written into *int c*.

The default value:

The fifth field is a string which contains an optional default value for the argument. It is defined as a string and automatically cast to the appropriate data type at run time. A NULL value indicates no default. In our example, **-tit** <**title**> and **y-int** have default values of "noname" and zero respectively, whereas **grad:gradient** has no default and is thus regarded as a mandatory argument.

The description string:

The sixth and final field allows the programmer to enter a brief description of the argument. It is these descriptions that are output by the arg_glossary() function. A NULL value causes that entry to be omitted from the glossary output.

7.3 Parsing the command line.

Having defined the argument table, we can now use it to parse the command line arguments in argv[]. There are several ways to do this, but the simplest is to use the **arg_scanargv** function.

```
int arg_scanargv(int argc,
                 char** argv,
                 arg_rec *argtable,
                 int n,
                 char* CmdLine,
                 char* ErrMsg,
                 char* ErrMark);
```

It takes as input the command line arguments in **argv[ ]** (there are **argc** of them) and a pointer to the argument table in **argtable** (which has **n** rows). It proceeds to scan the argv[] arguments (skipping argv[0]) and tries to resolve them with the entries given in the argument table. If this can be done successfully then all argument values are written into the program variables as designated by the argument table and the function returns 1. If not, the function returns 0 to indicate failure.

The three string pointers **CmdLine**, **ErrMsg**, and **ErrMark** refer to user defined string buffers in which **arg_scanargv** returns information about the parsing. They are optional parameters in the sense that they may be given as NULL if you do not wish to use them.

CmdLine is always assigned a copy of the orginal command line, concatenated into a single space-separated string.

ErrMsg and **ErrMark** are only used when **arg_scanargv** detects a parse error in the command line. In those cases, **ErrMsg** is assigned an explantory error message string, and **ErrMark** is assigned a string of tilde characters which have been formatted in such a way as to highlight the exact location of the error in **CmdLine** when printed one atop the other.

The code fragment below demonstrates the use of **arg_scanargv**. It presumes that *argc*, *argv*, *argtable*, and *narg* are as defined in the example above.

```
{
char cmdline[200], errmsg[200], errmark[200];
```

```
if (!arg_scanargv(argc,argv,argtable,narg,cmdline,errmsg,errmark))
   {
   // arg error occurred, print error message and exit
   printf("ERROR: %s\n", cmdline);
   printf("       %s %s\n", errmark, errmsg);
   return 1;
   }

// only get here if the arguments were scanned successfully

}
```

And here are some examples of the console output that this code produces when **arg_scanargv** detects a parse error.

```
$ myprog grad:oops
ERROR: myprog grad:oops
                   ^ invalid grad:gradient argument
$ myprog grad:13 nope
ERROR: myprog grad:13 nope
                      ^^^^ unexpected argument
$ myprog grad:13 99 uh oh
ERROR: myprog grad:13 99 uh oh
                         ^^ unexpected argument
```

7.4 Generating online help.

The arg_syntax() and arg_glossary() functions take an argument table and generate plain text descriptions of its command line syntax as well as descriptions of the individual arguments. These are useful for displaying help text to the user.

```
const char* arg_syntax(const arg_rec* argtable, int n);
const char* arg_glossary(const arg_rec* argtable, int n, const char* prefix);
```

The **arg_syntax** function returns a pointer to an internal string buffer that contains a plain text description of the usage syntax of the argument table it was passed. The string comprises a space separated list of the tag and name strings of each argument table entries concatenated into a single line string. Optional command line arguments are automatically enclosed in square brackets. Calling **arg_syntax** on our example argument table would return the string:

```
[-tit <title>] grad:gradient [y-int]
```

The calling program would ordinarily prepend the program name from argv[0] to this to get the full command line usage syntax.

The **arg_glossary** function is similar, except it generates a multi-line text string with one argument per line. Each line includes the argument's tag, its name string, and its description string as given in the argument table. Arguments that have a NULL description string are omitted. Each line of the glossary string is prefixed with the string given in the prefix parameter; it useful for indenting each line of the string. Calling **arg_glossary** with our example argument table results in the following multi-line string:

```
-tit \<title\> your title
grad:gradient line gradient
y-int y-intercept
```

7.5 Putting it all together.

Lets return to our example program and put it all together in its entirety. Our program, when executed without any arguments (argc==1), will print the argument usage syntax and a glossary on stdout, then exit.

When given a valid set of arguments, it will display the resulting argument values as they are stored in the local program variables. Otherwise, it echoes the erroneous command line together with an appropriate error message and terminates with error code 1.

```
#include "argtable.h"

int main(int argc, char **argv)
   {
   static char str[50];
   static double grad;
   static int c;
   arg_rec argtable[] =
      {
      {"-tit ", "<title>",  arg_str, str,   "noname", "\t\t your title"},
      {"grad:", "gradient", arg_dbl, &grad, NULL,     "\t line gradient"},
      {NULL,    "y-int",    arg_int, &c,    "0",      "\t\t y-intercept"}
      };
   const size_t narg = sizeof(argtable)/sizeof(arg_rec);

   // process the command line args
   if (argc==1)
      {
      // display program usage and exit.
      printf("Usage: %s %s\n", argv[0], arg_syntax(argtable,narg));
      printf("%s\n",arg_glossary(argtable,narg,"  "));
      return 0;
      }
   else
      {
      // scan command line arguments from argv[]
      char cmdline[200], errmsg[200], errmark[200];
      if (!arg_scanargv(argc, argv, argtable, narg, cmdline, errmsg, errmark))
         {
         // arg error occurred, print error message and exit
         printf("ERROR: %s\n", cmdline);
         printf("       %s %s\n", errmark, errmsg);
         return 1;
         }
      }

   // get here only if command line args ok
   printf("title: \"%s\"\n",str);
   printf("gradient: %f\n",grad);
   printf("y-intercept: %d\n",c);

   return 0;
   }
```

Here are some results of running myprog with various command line arguments.

```
$ myprog
Usage: myprog [-tit <title>] grad:gradient [y-int]
  -tit <title>     your title
  grad:gradient    line gradient
  y-int            y-intercept

$ myprog grad:10
title: "noname"
gradient: 10.000000
y-intercept: 0

$ myprog 7 grad:1.234
title: "noname"
```

```
gradient: 1.234000
y-intercept: 7

$ myprog -tit "hello world" grad:3.33 11
title: "hello world"
gradient: 3.330000
y-intercept: 11
```

7.6 Release Notes.

Argtable-2.0 coming eventually.

Argtable-2.0 will be a major overhaul of the code. The changes are required to address the most common complaint about argtable; the potential for buffer overruns as argtable writes into fixed size string buffers. The redesign will bring some inevitable changes to the library interface, but the basic look and feel of the argument tables will stay the same. I been promising to finish Argtable-2.0 for a very long time. Well, one of the days....

Argtable-1.3 released December 20, 2001.

Moved argtable to a new home on the **sourceforge** site and revamped the documentation. Documentation is now created with **Doxygen** instead of **c2man**. Also fixed some minor bugs in the Makefiles. The source code itself is unaltered.

Argtable-1.2 released August 5, 1999.

The original makefiles have been replaced by autoconf makefiles. The char pointers in the argument table have been redefined as pointers to const char. Some of argtable's internal string buffers have been made larger to accommodate long command lines, and a bug that occurred when program names contained whitespace has been fixed. The documentation has also been revised.

Argtable-1.1 released January 20, 1999.

This version fixes some cross-platform compilation errors, and saw the introduction of the multi-platform configuration. It also saw the addition of the arg_record() function and a change to the arg_scanargv() function so that it no longer requires argv[0] to be the first entry of the argument table. To maintain backwards compatibility, programs written for version 1.0 should now define the macro define ARGTABLE_COMPATIBILITY_10 prior to including the argtable.h header file.

Argtable-1.0 released November 13, 1998.

Argtable's debut!

Index

arg_bool
    argtable.h, 7
arg_catargs
    argtable.h, 9
arg_dbl
    argtable.h, 7
arg_dump
    argtable.h, 10
arg_glossary
    argtable.h, 9
arg_int
    argtable.h, 7
arg_lit
    argtable.h, 7
arg_rec, 4
    argdescrip, 6
    argname, 5
    argtype, 5
    defaultstr, 5
    tagstr, 5
    valueptr, 5
arg_record
    argtable.h, 9
arg_scanargv
    argtable.h, 7
arg_scanstr
    argtable.h, 8
arg_str
    argtable.h, 7
arg_syntax
    argtable.h, 8
arg_type
    argtable.h, 7
arg_typestr
    argtable.h, 10
argdescrip
    arg_rec, 6
argname
    arg_rec, 5
argtable.h, 6
    arg_bool, 7
    arg_catargs, 9
    arg_dbl, 7
    arg_dump, 10
    arg_glossary, 9
    arg_int, 7
    arg_lit, 7
    arg_record, 9
    arg_scanargv, 7
    arg_scanstr, 8
    arg_str, 7
    arg_syntax, 8
    arg_type, 7
    arg_typestr, 10
    ARGTABLE_VERSION, 7
ARGTABLE_VERSION
    argtable.h, 7
argtype
    arg_rec, 5

defaultstr
    arg_rec, 5

tagstr
    arg_rec, 5

valueptr
    arg_rec, 5